# Techniques d'aspect pour la gestion de la mémoire répartie dans un environnement CORBA/C++

T. Soueid[1,2], N. Yahiaoui[1,3], L. Seinturier[2], B. Traverson[1]
*1. EDF R&D 1 avenue du Général de Gaulle F-92140 Clamart France*
*2. Université de Paris 6, Laboratoire LIP6 8 rue du Capitaine Scott F-75015 Paris France*
*3. Université de Versailles, Laboratoire Prism 45 avenue des Etats-Unis F-78035 Versailles France*

***Résumé*—La gestion de la mémoire répartie dans les environnements CORBA C++ n'est pas standardisée. Non seulement elle doit être prise en charge par l'application alors qu'il s'agit d'une préoccupation plutôt technique mais, de plus, elle est transverse à l'ensemble de l'application. Elle doit être mise en place dès que les objets fonctionnels de l'application sont à la fois volumineux et utilisés de façon transitoire. Elle impacte à la fois le côté client CORBA qui utilise une référence vers un objet distant et le côté serveur CORBA qui gère un compteur de références. La programmation orientée aspect apporte une solution permettant de tisser la gestion de mémoire répartie au sein d'une application CORBA. Cet article rend compte de la mise en œuvre de l'intégration de la gestion de la mémoire répartie dans un environnement scientifique fondé sur CORBA en C++ avec l'outil AspectC++, tisseur statique de C++ dont il a fallu adapter l'utilisation à un environnement CORBA.**

## I. Introduction

Pour ses besoins de simulation en environnement scientifique, EDF R&D utilise une plate-forme répartie qui repose sur l'architecture CORBA (Common Object Request Broker Architecture) [3]. Lors d'une simulation, des milliers d'objets CORBA peuvent être créés. Il devient alors indispensable de pouvoir gérer correctement la création et la destruction de ces objets en mémoire vive afin que la plate-forme puisse continuer à fonctionner correctement.

Ce problème a été identifié par l'OMG (*Object Management Group*), organisme qui conçoit et gère les spécifications CORBA. Il est pris en charge dans le service de gestion de cycle de vie (*Life Cycle Service*) [6]. Cependant, cette spécification n'est qu'un ensemble de recommandations et l'implantation est laissée au développeur de l'application répartie qui possède un libre choix d'implanter ce mécanisme de la façon la plus adaptée à son application.

Le cycle de vie dans CORBA est du point de vue des utilisateurs, des concepteurs et des développeurs des applications scientifiques un aspect purement technique. La séparation du fonctionnel et du technique est un sujet d'intérêt actuel dans la recherche en informatique. Les aspects techniques d'une application sont en général des préoccupations transverses à l'ensemble de l'application. Elles sont de ce fait éparpillées un peu partout dans le code source de l'application. La programmation orientée aspect est une approche permettant la modularisation de ces préoccupations transverses [5].

Notre objectif est d'étudier l'application de la programmation orientée aspect - dans un cadre C++ / CORBA en particulier - dans le contexte de notre problématique de gestion de cycle de vie des objets CORBA.

Cet article est constitué de trois parties. La première partie rappelle les principes de la programmation orientée aspect et leur mise en œuvre dans l'outil Aspect-C++ [1][7]. La deuxième partie s'intéresse à la gestion de la mémoire répartie dans CORBA. La troisième partie en présente l'aspectualisation.

## II. Programmation orientée aspect

Les techniques de programmation ont évolué de concert avec celles des systèmes informatiques. Ces systèmes ont, en effet, tendance à devenir de plus en plus complexes. La programmation orientée objet (POO) a vu le jour suite à la constatation que les systèmes, devenus de plus en plus complexes, pouvaient être exprimés en terme d'un ensemble d'objets – plus simples - qui interagissent.

La POO est l'approche de choix pour la plupart des projets logiciels actuels. Elle est particulièrement efficace dans l'expression des fonctionnalités dites verticales, capacités fonctionnelles exprimant les aspects métiers du système. Par contre, elle s'avère particulièrement limitée dans l'expression des fonctions dites horizontales ou transverses, celles exprimant les aspects techniques de l'application. En effet, de telles capacités techniques tendent à s'étaler sur un grand nombre d'objets souvent sans rapport entre eux.

La programmation orientée aspect (POA) a été conçue afin

de pallier cette faiblesse de la POO dans l'expression des fonctions transverses. Nous allons rappeler, dans cette partie, les principes de la POA puis en présenter une mise en œuvre outillée dans un environnement C++.

### A. *Principes*

Un système logiciel peut être considéré comme un ensemble de préoccupations métier (fonctionnelles) et techniques. Les capacités techniques sont, par exemple, la persistance des données, la sécurité, la journalisation, le traitement des exceptions … Ces différentes fonctions sont intégrées dans les modules composant le système (voir Figure 1).

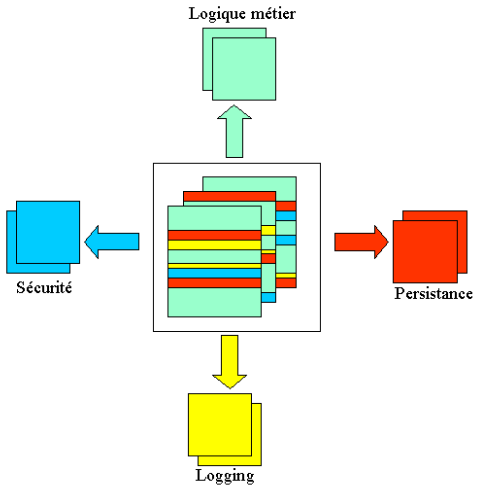


Fig. 1 – Un système vu comme un ensemble de préoccupations

La programmation orientée aspect est une approche permettant de rendre modulaire l'implantation des préoccupations transverses, de les implanter indépendamment les unes des autres et de les combiner ultérieurement pour produire le système final. L'unité de modularité en POA est appelée un *Aspect* tout comme l'unité de modularité en POO est appelée une *Classe*.

Plus techniquement, un **aspect** est une unité de regroupement d'une ou de plusieurs définitions de coupe, d'une ou de plusieurs définitions de conseil ou d'une ou de plusieurs associations de coupes à des conseils. Un **conseil** (*advice* en anglais) est un fragment de code à insérer au niveau de points de jonction et qui implante une préoccupation transverse. Un **point de jonction** (*joinpoint* en anglais) est un endroit précis dans l'exécution du programme (par exemple, un appel à une méthode, à un constructeur …). Une **coupe** (*pointcut* en anglais) constitue le moyen de spécifier un ensemble de points de jonction particuliers. Une coupe est souvent exprimée sous forme d'une expression régulière. Un **tisseur** (*weaver* en anglais) est un outil spécial permettant d'appliquer les aspects au code de base.

La définition et l'implantation exacte de chacun des concepts introduits ci-dessus dépendent bien évidemment de la spécificité du langage ou du cadre logiciel utilisé. Certaines subtilités (dans la définition des aspects par exemple) peuvent en effet exister entre différents langages de POA.

### B. *AspectC++*

Les concepteurs d'AspectC++ fournissent un utilitaire permettant de tisser des aspects dans du code C++ [1]. La figure suivante illustre le principe de fonctionnement de ce tisseur (voir Figure 2).

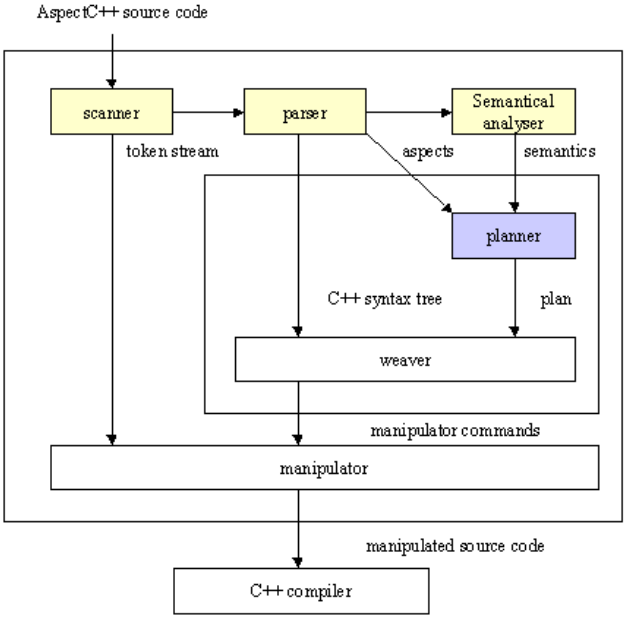


Fig. 2 – Principe du fonctionnement du tisseur AspectC++

Dans un premier temps, le code source C++ est décomposé puis analysé. Dans un second temps, une phase de planification permet d'évaluer les expressions de coupe et d'évaluer les ensembles de points de jonction concernés. Un plan est établi pour le tisseur indiquant quel code insérer au niveau de chaque point de jonction. Dans un troisième temps, le tisseur établit d'une façon concrète selon l'arbre syntaxique C++ généré précédemment comment tisser le code des aspects en respectant la syntaxe et la sémantique C++. Enfin, la manipulation du code C++ initial est effectuée par le manipulateur de code qui produit le code C++ modifié incluant le code des aspects. Ce code ne contient aucune construction AspectC++ mais du code C++ ordinaire qu'on pourra compiler avec n'importe quel compilateur C++.

## III. GESTION DE MÉMOIRE RÉPARTIE CORBA

Comme nous l'avons indiqué en introduction, il est nécessaire que chaque application répartie fournisse sa propre implantation du *Life Cycle Service* des objets CORBA. Nous en proposons une mise en œuvre fondée sur trois entités : les Clients, les Fabriques et les Objets.

Nous détaillons dans cette partie les principes de notre mécanisme et nos choix d'implantation en environnement CORBA.

### *A. Principes*

L'objet dont on souhaite gérer le cycle de vie, doit effectuer un comptage des références que les clients détiennent sur lui. Un compteur de références est incrémenté à chaque fois qu'un client s'enregistre auprès de l'objet. En effet, un client peut demander de créer un objet ou de récupérer une référence sur un objet déjà existant et, dans les deux cas, l'objet incrémente la valeur d'un compteur interne.

Lorsqu'un client n'a plus besoin d'utiliser un objet, il demande de libérer sa référence sur l'objet. L'objet procède ainsi à la décrémentation de son compteur de références. Lorsque ce compteur devient égal à zéro, cela indique que plus aucun client n'utilise l'objet et il peut ainsi être détruit.

La demande de création ou de récupération d'un objet se fait auprès d'une fabrique (*Factory*) ayant la responsabilité de ce type d'objet. C'est donc la fabrique qui informe l'objet qu'un nouveau client souhaite s'enregistrer afin qu'il incrémente son compteur. Par contre, pour libérer un objet, le client peut appeler directement la méthode de libération de l'objet.

### *B. Mise en œuvre CORBA*

Dans le cadre de notre mise en œuvre CORBA, les objets peuvent être accédés par plusieurs clients et traiter plusieurs requêtes simultanément (*multi-threading*). Nous sommes dans une situation d'accès partagé aux objets CORBA. Ainsi, un objet ne devra pas être détruit tant qu'il y a au moins un client qui l'utilise ou une activité en cours d'exécution. Nous avons donc deux niveaux de contrôle dans la gestion du cycle de vie d'un objet : gestion des activités simultanées et gestion de plusieurs clients. Le premier niveau de contrôle est standardisé par l'OMG. Le deuxième niveau de contrôle a dû être ajouté dans le cadre de notre projet.

#### *1) Gestion des activités simultanées*

Par défaut, le *Portable Object Adapter* (POA[1]) maintient une table des objets CORBA actifs (*Active Object Map*). Elle contient les associations entre l'identificateur de l'objet CORBA et le servant qui l'incarne. Le POA utilise ces associations afin de localiser le servant correspondant à un objet CORBA [4].

Afin d'éviter que les servants ne consomment inutilement de la mémoire, il est nécessaire de les détruire lorsqu'ils ne sont plus utilisés. Le principe serait de désactiver l'objet CORBA concerné (appel à `deactivate_object`) et de détruire le servant qui l'incarne (appel à `delete`).

Cependant, dans le cadre d'une application *multi-thread*, un servant peut traiter plusieurs requêtes simultanément. Cette situation implique que la libération d'une référence sur un objet ne devrait pas déclencher sa destruction tant qu'une activité est en cours. Un mécanisme de comptage de références est alors nécessaire permettant d'indiquer, à un moment précis, le nombre d'activités en cours sur un objet. Ainsi, un servant ne peut pas être détruit par un *thread* s'il est toujours utilisé par un autre *thread*.

Ce niveau de contrôle est fourni en standard dans l'architecture CORBA. Il est en effet possible de faire dériver les servants de la classe `RefCountServantBase`. Cette classe fournit des implantations de `_add_ref` et de `_remove_ref` permettant d'effectuer un comptage de références pour les classes dérivées. L'implantation de `_add_ref` incrémente le compteur de références. L'implantation de `_remove_ref` décrémente le compteur et appelle `delete` sur `this` quand ce compteur devient égal à zéro. Le POA invoque `_add_ref` à chaque fois qu'une requête est invoquée sur le servant depuis un *thread* et appelle `_remove_ref` quand l'exécution est terminée (voir Figure 3).

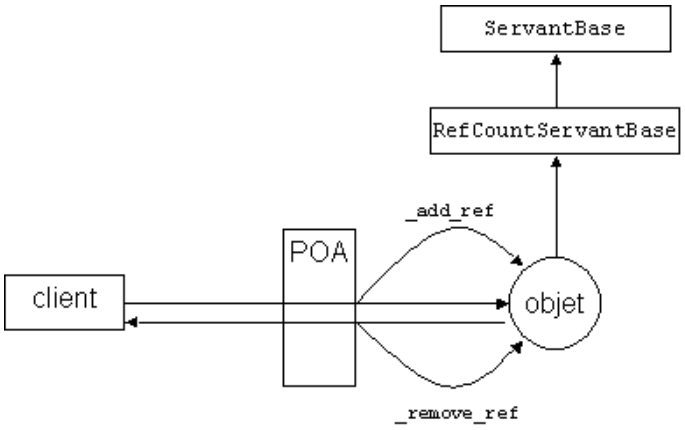


Fig. 3 – Solution permettant de gérer des activités simultanées

#### *2) Gestion de plusieurs clients*

En ce qui concerne l'utilisation d'un objet CORBA par plusieurs clients simultanément, il n'existe aucun mécanisme standard permettant de compter le nombre de clients et de détruire l'objet lorsqu'il n'est plus utilisé. L'application devra alors gérer cela de la façon la plus adéquate.

Dans le cadre de notre projet, un mécanisme additionnel de comptage de références est employé. Ce second compteur de références permet d'indiquer le nombre d'objets clients qui détiennent des références sur l'objet et donc qui utilisent toujours les services fournis par l'objet. L'interface de l'objet expose deux méthodes supplémentaires. Une première méthode permet d'ajouter un nouveau client et d'incrémenter le compteur de l'objet. Une autre méthode permet de retirer un client et de décrémenter le compteur. Si le compteur devient égal à zéro, l'objet sait qu'il n'est plus utilisé par aucun client et

[1] Dans cette section et dans la figure 3, POA désigne l'adaptateur d'objets de l'architecture CORBA et non pas, évidemment, la Programmation Orientée Aspect.

peut ainsi déclencher sa destruction (voir Figure 4).

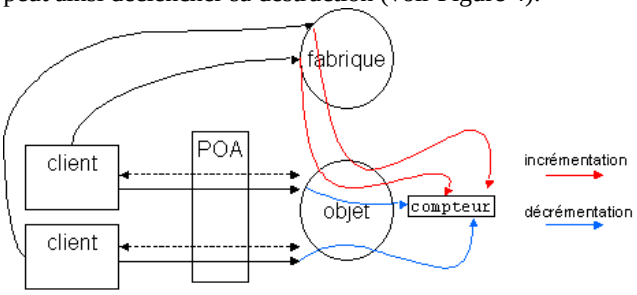


Fig. 4 – Solution permettant de gérer plusieurs clients

L'objet CORBA possède ainsi deux compteurs différents. Le premier est un compteur de références hérité de `RefCountServantBase` qui indique le nombre de références actives à un moment donné sur l'objet et permet donc de gérer les activités simultanées. Le second est un compteur de clients qui indique à un moment donné le nombre de clients qui détiennent des références sur l'objet et qui utilisent ses services. Ce second compteur permet de gérer le partage de l'objet entre plusieurs clients. L'exemple ci-dessous illustre le fonctionnement de ce second compteur et en montre un cas d'utilisation sur une maquette (voir Figure 5).

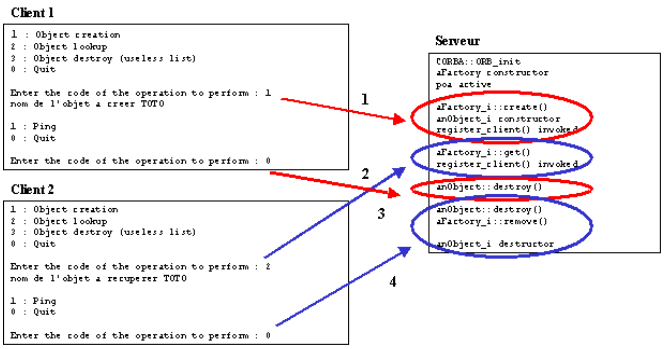


Fig. 5 – Fonctionnement sur une maquette

Dans l'exemple ci-dessus, deux clients se connectent à un serveur. Le serveur permet la création de nouveaux objets, la récupération de références sur des objets existants et la destruction d'objets qui ne sont plus utilisés.

Dans notre scénario, un premier client demande au serveur la création d'un nouvel objet en invoquant la méthode `create` de la fabrique (étape 1). Le client est alors enregistré par la fabrique auprès de l'objet en invoquant `register_client`. Un autre client demande la récupération d'une référence vers cet objet en invoquant `get` sur la fabrique (étape 2). Ce client est à son tour enregistré par la fabrique auprès de l'objet. Dans la troisième étape, le premier client libère sa référence sur l'objet en invoquant la méthode `destroy` de l'objet (étape 3). Cependant, l'objet n'est pas détruit car il sait qu'un autre client détient toujours un référence sur lui. C'est dans l'étape 4 que le second client demande de libérer l'objet et que ce dernier est effectivement détruit.

## IV. Aspectualisation

La gestion du cycle de vie des objets CORBA est un aspect purement technique et une préoccupation transverse à l'ensemble des objets fonctionnels. Dans une plate-forme très complexe, introduire un mécanisme de gestion du cycle de vie des objets CORBA devient un travail conséquent car il faudrait modifier différents types d'objets dont les définitions sont éparpillées dans une multitude de fichiers source. Même en acceptant d'effectuer ce travail manuellement, il est clair que toute modification qu'on souhaiterait apporter ultérieurement au niveau du mécanisme impliquerait d'aller modifier tous les fichiers concernés. Il faut alors s'assurer de répercuter la modification au niveau de tous les objets concernés.

Puisque notre plate-forme scientifique est codée en C++, le langage orienté aspect le plus adapté dans ce cas est AspectC++ que nous avons introduit précédemment.

Le mécanisme proposé implique, comme nous l'avons vu dans la partie précédente, trois types d'entités : les objets CORBA, les fabriques et les clients. Il implique une modification au niveau de ces trois entités. Nous détaillons par la suite, différentes possibilités d'utilisation de la programmation orientée aspect au niveau de ces trois entités.

### A. *Aspectualisation des objets*

Comme nous l'avons vu, il s'agit d'ajouter aux objets un mécanisme de comptage de références. Les modifications apportées aux objets sont l'ajout de deux compteurs (un compteur de références en étendant la classe `RefCountServantBase` et un compteur du nombre de clients qui détiennent des références sur l'objet) et de deux méthodes permettant l'enregistrement et le retrait d'un client auprès de l'objet. De plus, l'objet dépend maintenant d'une fabrique qui le crée et qui l'informe de l'enregistrement de nouveaux clients.

Toutes ces modifications peuvent faire l'objet d'un aspect (« Aspect GM » dans la Figure 6) en AspectC++ qui modifie l'objet de base :

- Ajout de l'héritage de `RefCountServantBase`,
- Introduction d'un compteur de clients (une variable de type entier),
- Introduction des deux méthodes `register_client()` et `destroy()`,
- Introduction d'une référence vers la fabrique.

La figure suivante (Figure 6) illustre ce principe. A gauche, la classe « an_object » avant tissage de l'aspect « GM » possède ses méthodes propres « method1 » et « method2 ». A droite, la classe « an_object » après tissage de l'aspect « GM » inclut les modifications indiquées précédemment.

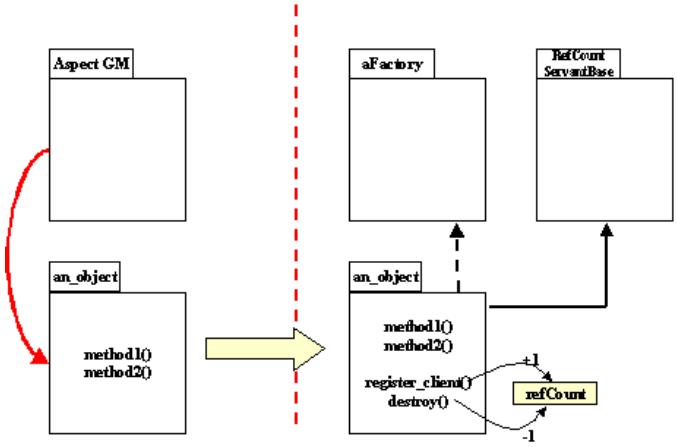


Fig. 6 – Modification des objets par un aspect

L'intérêt d'un aspect est de permettre, d'une part, de grouper ces modifications d'une façon modulaire dans une unique entité qu'est l'aspect. D'autre part, lorsque plusieurs types d'objets sont concernés par le mécanisme, l'aspect permettra d'éviter d'aller modifier chaque objet à part mais plutôt de tisser ces modifications en définissant la coupe adéquate.

### *B. Aspectualisation des fabriques*

Les fabriques sont des entités qui sont introduites par le mécanisme de gestion mémoire et qui gèrent la création des objets et la récupération de références par les clients.

Si l'application n'est pas déjà équipée de ce type d'entités, elles constituent de nouvelles classes qui seront introduites par le mécanisme. Il n'est pas nécessaire d'utiliser les aspects dans ce cas.

Si les fabriques existent déjà au niveau de l'application, il faut les modifier pour qu'elles prennent en compte le mécanisme de gestion de cycle de vie.

Dans les deux cas, il faut introduire dans la fabrique les deux méthodes permettant la création (`create`) d'un nouvel objet et la récupération (`get`) d'une référence vers un objet existant.

### *C. Aspectualisation des clients*

Au niveau des clients, il faut désormais passer par une fabrique pour chaque type d'objet qu'on souhaite utiliser. Il faut aussi s'assurer que le client s'enregistre auprès d'un objet à chaque fois qu'il acquiert une référence vers cet objet. D'un autre côté, il devra informer l'objet lorsqu'il libère sa référence vers cet objet.

L'aspect devra apporter les modifications suivantes au niveau du code du client :

- Récupérer une référence sur une fabrique adéquate pour chaque type d'objet. Cela pourra se faire au début de l'initialisation ou du démarrage du client.
- Invoquer la méthode `create()` ou `get()` avant l'utilisation d'un objet. `create()` sera appelée si le client crée un nouvel objet et `get()` sera appelée si l'objet existe déjà (a été crée par un autre client) et que le client souhaite récupérer une référence vers cet objet.
- Invoquer `destroy()` lorsque le client n'utilise plus les services de l'objet et qu'il souhaite libérer toutes ses références sur cet objet.

Dans le cadre de notre projet, cette partie a été réalisée manuellement. L'automatisation de la modification des clients repose sur des techniques d'analyse de code qui sortent du cadre aspect.

## V. CONCLUSION

Nous avons présenté, dans cet article, la problématique du cycle de vie dans un environnement CORBA et une solution fondée sur la programmation orientée aspect dans un cadre C++.

L'utilisation des aspects dans un cadre purement C++ est assez rare au niveau de l'offre actuelle d'outils. Dans le cadre de notre projet fondé sur les choix de CORBA et de C++, nous avons rencontré des difficultés liées au tissage d'aspect. Par exemple, une de ces difficultés est l'introduction de méthodes lorsqu'elles ont été aussi déclarées au niveau de l'IDL CORBA. En effet, les méthodes déclarées au niveau IDL se trouvent aussi déclarées au niveau du squelette du servant CORBA. Le tisseur d'AspectC++ refuse alors d'introduire ces méthodes au niveau de la classe du servant concerné. Nous devons alors forcer le tisseur, à travers plusieurs options, à introduire ces modifications au niveau du code.

Au niveau fonctionnel, l'outil AspectC++ est toujours en cours d'évolution. Dans la version actuelle, plusieurs capacités sont en cours de développement telle que la déclaration de coupes sur les constructeurs et les destructeurs ou l'introduction de nouveaux constructeurs au niveau d'une classe.

Enfin, dans la version actuelle d'AspectC++, le code source C++ modifié à la sortie du tisseur est très difficile à analyser du fait que le tisseur ne respecte aucune règle de lisibilité du code. Récemment, les concepteurs d'AspectC++ ont fourni un *plugin* permettant le support du langage dans l'environnement de développement *Eclipse*. Cet outil permet une visualisation plus intuitive et graphique des aspects et de leur interaction avec le code de base et permet aussi une gestion plus automatisée de la génération d'application à travers la génération automatique de fichiers de *Make*.

Nous allons, pour finir, présenter quelques évolutions qui nous ont paru intéressantes à prendre en considération suite à cette expérimentation.

Tout d'abord, AspectC++ ne fournit pas un cadre ou des mécanismes permettant de tisser des aspects à l'exécution. Néanmoins, une forme de dynamicité peut être réalisée à travers un mécanisme de réflexivité. Il est en effet possible d'obtenir, à tout moment du programme, des informations sur le point de jonction actuel à travers la référence

`thisJoinPoint` qui pointe vers l'objet actuel. Les méthodes fournies par `thisJoinPoint` permettent d'accéder à des attributs statiques du point de jonction (tel que sa représentation sous forme de chaîne de caractères, son identificateur, ses types d'arguments …), mais aussi à des attributs dynamiques (tel que les valeurs actuelles des arguments d'un point de jonction d'appel de méthode). Ce mécanisme permet d'implanter des aspects génériques qui peuvent manipuler des structures dont le type est inconnu à la compilation. Enfin, AspectC++ pourrait s'appuyer sur les extensions RTTI (*Run Time Type Information*) de C++ ou sur OpenC++ [2] (boîte à outils ou *toolkit* de méta-programmation pour C++) afin de fournir un cadre permettant un tissage dynamique.

Ensuite, au niveau de CORBA, nous n'avons aucun moyen pour indiquer la nature distincte introduite par notre mécanisme. Il serait intéressant si, au niveau de l'IDL CORBA, nous pouvions préciser la nature transverse de notre mécanisme. En d'autres termes, il faudrait étendre les concepts de la programmation orientée aspect au niveau indépendant du langage de programmation. Cela nécessiterait des changements au niveau IDL où nous ne travaillons qu'en termes d'interfaces et de méthodes afin d'introduire les notions d'aspects, de points de jonction, de coupes et de conseils. Un telle approche permettrait d'abord de générer une bonne partie du code des aspects d'une façon automatisée et dans plusieurs langages cible (Aspect C++ pour C++ et AspectJ pour Java).

Enfin, la modélisation orientée aspect présente les mêmes bénéfices à la phase de conception que ceux que la programmation orientée aspect présente à la phase d'implantation. Ainsi, supporter les aspects en tant que constructions au niveau de la conception permettrait au développeur de mieux comprendre et documenter son modèle, ainsi que de réutiliser ou changer les composantes de ce modèle d'une façon plus facile. Le domaine de la modélisation par aspects est encore embryonnaire avec une multitude d'équipes de recherche en informatique qui proposent leurs notations et leur vision des choses. L'ultime objectif est d'aboutir à une notation unifiée qui sera intégrée aux ateliers de conception logiciels et qui permettra d'aborder les aspects dans les premières phases du cycle de développement d'un logiciel.

## BIBLIOGRAPHIE


[1] http://www.aspectc.org

[2] Chiba S., **"A Metaobject Protocol for C++"**, Proceedings of OOPSLA'95, ACM SIGPLAN Notices 30(10): 285-299, October 1995, ACM Press.

[3] http;//www.omg.org

[4] Henning M. et al., **"Advanced CORBA Programming with C++"**, Addison-Wesley Professional Computing Series.

[5] Kiczales G. et al., **"Aspect-Oriented Programming"**, ACM Computing Surveys, 28(4), December 1996.

[6] Object Management Group, **"Life Cycle Service Specification"**, www.omg.org.

[7] Spinczyk O. et al., **"AspectC++: An Aspect-Oriented Extension to C++"**, Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems.